SageMath introduction

Application of discrete models

SageMath (in short, Sage) is a computer algebra system, which covers different areas of mathematics (algebra, calculus, number theory etc.). It is free software ("free" as in "freedom", licensed under the GNU GPLv3), e.g. it doesn't restrict the user in using the software.
The language of Sage is based on the Python programming language. Sage is built on top of many other more specialized mathematical software (NumPy, SymPy, Maxima, GAP, R etc.), and it provides a common interface for them, extending them with its own functionality.
The website of Sage: https://www.sagemath.org/

1 Accessing SageMath

The two main ways of accessing Sage is to either download and install it on a computer, or to connect to a Sage server through the internet.
Installing Sage is not trivial, because it is a very complex software package. However, after doing it once, it's easier to use, faster, more reliable (you don't need an internet connection), and you have complete control over your computations.
Using an online Sage server is easier for the first time, but it suffers from the same problems as any other online service: you need constant internet connection to use it, that particular server must be always available, and you don't have control over your computations (you can only hope that they don't modify or delete your files, or show them to someone you don't want to). (Remember: using an online service means just using someone else's computer.)

1.1 Installing Sage

Below is a simple explanation of some of the easiest ways to install Sage on your computer.
See the official description for all the possibilities:
https://doc.sagemath.org/html/en/installation/index.html

1.1.1 As root on Linux

The easiest way to install Sage is if you have administrative (*root*) priviliges on a Linux distribution that has Sage in its repositories (usually named `sagemath`). Then you can install it with one command, for example:

- Ubuntu, Debian: `sudo apt install sagemath`
- Arch Linux: `sudo pacman -S sagemath`
- Fedora: `sudo dnf install sagemath`

1.1.2 As a user on Windows/Linux

Otherwise, you need to find binary distributions of Sage. The newest versions (currently 10.4) don't have it, but slightly older versions (e.g. 9.3) do, for both Windows and Linux (and others). (These versions should be perfectly fine for our purposes.) They can be found on the following page:
https://www.sagemath.org/mirrors.html
Choose a mirror on this page (*mirror*: a copy of a website to reduce load on the original server, thus improving download speed). From the "Archived (Outdated)" section, choose the operating system (e.g. Windows or Linux), and download the appropriate (e.g. newest) file.
For Windows:

- Download the installer (an `.exe` file).
- Run it, and follow the instructions. (Note: you don't need to install the documentation, only the core package. Note 2: it asks you to agree to the license, which is nonsense, as Sage is free software, so it doesn't restrict you using the software whether you agree the license or not.)
- Run the created SageMath icon on the desktop. (First time it may take a while to start. Wait for the '`sage:`' prompt to appear.)

For Linux:

- Select your CPU architecture (most likely "64bit").
- Download the appropriate compressed archive. (They are given for Debian and Ubuntu, but they may also work in other distributions, possibly after some fiddling. University computers currently have Debian 11, but you can check it with the command '`cat /etc/os-release`'.)
- Extract the archive, e.g. by the console command '`tar -xf <filename>.tar.bz2`'. Note: on the university computers, your home directory (`~/`) won't have enough space for this package (it is 3GB compressed, and 12GB uncompressed), because it's on a network storage with quotas on it. You'll need to find a local place on the computer for which you have permissions (I found `/vms/`, which worked for me).
- Run the executable file `sage` from the extracted `SageMath` directory in a terminal. For example: '`cd <directory>/SageMath`', then run '`./sage`'). On the first run, it may take a while to start.

1.2 Online Sage

1.2.1 SageMathCell

A very simple Sage server can be found at https://sagecell.sagemath.org/.
Just type or copy the Sage code you want to run, and press Shift+Enter (or the Evaluate button) to see the result. If you want to save the code for later, you need to copy it to a text editor and save it on your computer (`*.sage`).

1.2.2 CoCalc

CoCalc, https://cocalc.com/, provides a feature-rich web-based interface to SageMath (and other software).
You need to sign up with an account to use it. Also, it has several restrictions on your account unless you pay.

2 Python basics

Since Sage is a Python-based language, first we have to learn Python.
Python is a general-purpose programming language, which allows quick development with its dynamic types and flexibility, but it is not the best choice in terms of efficiency.
Its website is https://www.python.org/, and its documentation is at https://docs.python.org/.

2.1 Some simple commands

Comments start with a hash sign (#):

```
# This is a comment.
```

The `print()` function can be used to print something:

```
print("Hello world!")
```

Variables don't need to be declared, only assigned:

```
a = 2+3*4
```

The assignment doesn't print anything, so print its value now:

```
print(a)
```

It can be seen that Python doesn't require semicolon (`;`) at the end of each statement (though it doesn't forbid it either). But it can be used to separate multiple statements in the same line:

```
print(a); print(a*2)
```

A number can be printed in multiple ways:

```
print("a =", a)
print("a = " + str(a))   # what happens without str()?
print("a = %d" % a)      # like printf() in C
print(f"a = {a}")
```

Note: further information can be requested by the `help(`*command*`)` command, e.g. `help(print)`.

2.2 Types

Python is a dynamically typed language, i.e. the types of the variables are detected at run-time when values are assigned to them, and they can be changed later with new assignments.
Some important types:

- `int`: integer, e.g. `123`
- `str`: string, i.e. sequence of characters, e.g. `"apple"`, `"123"` or `""` (empty string). Strings can be concatenated using the plus sign (`+`), e.g. `s = "app"; print(s + "le")`.
- `float`: floating-point number, e.g. `3.14`
- `bool`: logical (boolean) value, i.e. either `True` or `False`. The comparison relations give a `bool` result, e.g. `b = 3 > 2` is the same as `b = True` (see also the `if` statement later).
- `list`: list of elements, e.g. `[3, 1, 4, 1, 5]` (see later).

The type of a value can be queried by the `type()` function:

```
a = "apple"
print(type(a))           # <class 'str'>
print(type(12) == int)   # True
```

Values can be converted to different types by using the name of the new type as a function:

```
a = 12              # type(a) == int
print(a == "12")    # False (an int is never equal to a string)
a = str(a)          # type(a) == str
print(a == "12")    # True
```

2.3 Branching (`if`)

In Python, a branch can be written like this:

```
if a > 0:
   print("positive")
elif a == 0:
   print("zero")
else:
   print("negative")
```

We can see that in Python, the structure of the program is determined not by symbols (like `{` and `}` in C), but by *indentation*, i.e. by the spaces and tabulators in the beginning of the line. The head of the control structure is separated from its inner statement by a colon (`:`).

The control structure spans as long as the indentation is greater than at the beginning of the control structure. For example:

```
if a > 2:
   print("This is printed if 'a' is greated than 2.")
   print("Also this.")
if a <= 5:
   print("This is printed if 'a' is at most 5, regardless of the previous 'if'.")
   if a >= 1:
      print("This is printed if 'a' is between 1 and 5,")
      print("because this 'if' is inside the previous 'if' statement.")
   print("The inner 'if' ended, but the previous 'if' is still active,")
   print("so this is printed if a <= 5.")
print("This is always printed.")
```

(Attention: don't mix the spaces and the tabs! They might look similar to us, but not to the computer!)

A simple `if` statement can also be written in one line:

```
if a == 4: print("four")
```

The `if`'s condition be any logical expression (i.e. `bool` value). It can be constructed using relational operators (`<`, `>`, `<=`, `>=`, `==`, `!=`). Attention: the equality comparison is two equal signs (`if a == 5`), while the assignment is one (`a = 2`), don't mix the two. Logical expressions can be combined using the logical operators `and`, `or` and `not`, so for example the following statements are all equivalent:

```
if a >= 2 and a <= 5:     print("between 2 and 5")
if a >= 2 and not a > 5:  print("between 2 and 5")
if not (a < 2 or a > 5):  print("between 2 and 5")
```

If a variable is already a `bool`, then it doesn't need to be compared to `True` or `False`, it can be used directly in an `if`:

```
b = a > 10   # b is either True or False
if b: print("OK")         # this is preferred
if b == True: print("OK") # same, but unnecessary
# This is also possible (though not very useful):
if True: print("This is always printed.")
if False: print("This is never printed.")
```

More than that, Python allows not just a `bool`, but any expression inside an `if`, without any comparison. Then the condition means that the value is "not zero", "not empty" etc. For example:

```
a = 12  # or a = "abc", or a = [1,2]
if a: print("true")
if not a: print("false")
a = 0  # or a = "", or a = []
if a: print("false")
```

So '`if a`' is – depending on `a`'s type – the abbreviation of '`if a != 0`', '`if a != ""`', '`if len(a) != 0`' etc., while '`if not a`' is the opposite ('`if a == 0`' etc.).

2.4 Loops (`for`, `while`)

The following loop prints the numbers from 10 to 19:

```
for i in range(10, 20):
   print(i)
```

In Python, the ranges include the start element, but not the end, so the above loop goes only until 19, even though the endpoint was given as 20. (Note: this is consistent to the indexing of lists, where, as we will see, a list with 10 elements can be indexed from 0 to 9.)

This loop prints the squares of the given list element (9, 1, 16 etc.):

```
for i in [3,1,4,1,5,9]:
   print(i*i)
```

After the `in` keyword of the `for` statement, we can give any sequence-like object, e.g. a list, a string (which is a sequence of characters) or a generator (see the sequences later). One such sequence is `range()`, which generates an arithmetic series of integer, for example:

```
for i in range(2, 10): print(i)      # 2 3 4 5 6 7 8 9
for i in range(10): print(i)         # 0 1 2 3 4 5 6 7 8 9
for i in range(0, 10, 2): print(i)   # 0 2 4 6 8
for i in range(10, 0, -1): print(i)  # 10 9 8 7 6 5 4 3 2 1
for i in range(10, 0): print(i)      # prints nothing, because 10 >= 0
```

Loops can be nested, e.g. the following program prints a times table from 1 to 10:

```
for i in range(1, 11):
   for j in range(1, 11):
      print("%2d" % (i*j), end=" ")  # no line break after each number
   print()   # line break after a row of numbers
```

If the sequence contains composite elements, the loop variable can also be equally composite:

```
for (x,y) in [(1,2),(-2,3),(5,6)]:
   if x > 0:
      print(f"x={x}, y={y}")
```

The `while` loop runs as long as the condition is true. For example, the following loop is equivalent to a `for i in range(10): print(i)` loop:

```
i = 0
while i < 10:
   print(i)
   i += 1    # or: i = i + 1
```

The loop can contain `break` and `continue` statements like in other languages.
Note: Python has no post-test loops (like C's `do-while` loop).

2.5 Functions

Python functions can be defined as follows:

```
def add(a, b):
   return a + b
```

It can be used like this:

```
print(add(2, 3))
```

This is not very useful yet, so let's see something more complex:

```
def digits(n, base=10):
   dig = []
   while n > 0:
      dig.append(n % base)  # remainder
      n = n // base         # quotient
   dig.reverse()
   return dig
print(digits(127))     # [1, 2, 7]
print(digits(127, 2))  # [1, 1, 1, 1, 1, 1, 1]
print(digits(n=127, base=4))  # [1, 3, 3, 3]
```

Note: in Python, the body of a function (or a control structure like `if`) cannot be formally empty. To leave it empty (e.g. temporarily, if we want to write it later), we can use the `pass` command, which doesn't do anything:

```
def something(n):
   pass
something(2)   # no error, does nothing
```

Similarly:

```
if 3 == 3:
   pass
else:
   print("impossible")
```

2.6 Lists

One of the most important compound types of Python is `list`.
The simplest way to create a list is to give its elements between square brackets, e.g.: `[3,1,4,1,5]` or `[]` (empty list). In Python, lists can be inhomogenous, i.e. the elements don't need to have the same type, e.g.: `[1, "apple", False, [1,2,3], 123]`.
Lists can be converted from other sequences with the `list()` function:

```
print(list(range(5)))    # [0, 1, 2, 3, 4]
print(list("apple"))     # ["a", "p", "p", "l", "e"]
```

Lists can also be created by a formula using the `for-in` construction, similarly to a mathematical set-builder notation (e.g. $\{x^2 \mid x \in \{0, \ldots, 4\}\}$):

```
print([i*i for i in range(5)])  # square numbers from 0 to 16: [0, 1, 4, 9, 16]
print([i*i for i in range(10) if i%2 != 0])  # odd square numbers from 1 to 81
print([2*i for i in [3,1,4,1,5]])  # [6,2,8,2,10]
print([[i*j for i in range(1,6)] for j in range(1,6)])  # times table with lists
```

The list elements can be queried or changed by the square bracket notation:

```
L = [3,1,4,1,5,9]
print(L[0])   # 3   (Python counts from 0)
print(L[1])   # 1
print(L[4])   # 5
print(L[6])   # IndexError: list index out of range
print(L[-1])  # 9   (counting backwards)
print(L[-2])  # 5
L[2] = 6
print(L)      # [3,1,6,1,5,9]
L[10] = 10    # IndexError: list assignment index out of range
```

The number of elements in a list can be queried by the `len()` function, so for example the following two loops are equivalent:

```
L = [3,1,4,1,5,9]
for elem in L: print(elem)
for i in range(len(L)): print(L[i])
```

Elements can be added to or removed from a list:

```
L = [3,1,4,1,5]
L.append(9)     # insert 9 at the end
print(L)        # [3,1,4,1,5,9]
L.pop()         # remove the last element (9)
print(L)        # [3,1,4,1,5]
L.pop(2)        # remove L[2] (== 4)
print(L)        # [3,1,1,5]
L.insert(1, 9)  # insert 9 as the second element (L[1])
print(L)        # [3,9,1,1,5]
```

(See also: `help(list)`.)

Lists can be concatenated and multiplied:

```
L1 = [3,1];  L2 = [4,1]
print(L1 + L2)  # [3,1,4,1]
print(L1 * 3)   # [3,1,3,1,3,1]
print([2] * 5)  # [2,2,2,2,2]
```

Lists can be *sliced*, i.e. a range of elements can be queried:

```
L = [3,1,4,1,5,9]
print(L[3:5])  # [1,5]   (from L[3] up to but not including L[5])
print(L[3:])   # [1,5,9] (from L[3] to the end)
print(L[:3])   # [3,1,4] (all elements until L[3], not including that)
print(L[:])    # [3,1,4,1,5,9] (all elements, i.e. the copy of L, see below)
print(L[3::2]) # [1,9] (from L[3], every second element)
print(L[::-1]) # [9,5,1,4,1,3] (all elements backwards)
L[-4:-1] = [2,2] # replace 3 elements from L[-4] with two elements: [2,2]
print(L)       # [3,1,2,2,9]
```

In Python, if a list (or other compound object) is assigned to another variable, it is not copied, but the same list will be referenced by both variables. To actually copy a list, use the `L[:]` construct. For example:

```
L1 = [3,1,4]
L2 = L1     # not a copy, but L1 and L2 refer to the same list
L2[1] = 2   # both are changed!
print(L1)   # [3,2,4]
print(L2)   # [3,2,4]
L3 = L1[:]  # really a copy, i.e. there will be two lists
L3[0] = 1   # only L3 is changed
print(L1)   # [3,2,4]
print(L3)   # [1,2,4]
```

2.7 Other sequences

A `tuple` is very similar to a list, but it cannot be modified after creation (i.e. it is *immutable*). Otherwise it works in the same way, except that it is created using round brackets:

```
T = (3,1,4)
print(T[-1])      # 4
print(T[1:])      # (1,4)
print(T + (1,5))  # (3,1,4,1,5)
print(2*T)        # (3,1,4,3,1,4)
T[2] = 6          # TypeError: 'tuple' object does not support item assignment
```

A string (`str`) is a sequence of characters. It can be given between quotes, in multiple ways (the first two lines are equivalent):

```
print("I'm OK.")
print('I\'m OK.')
```

```
print("""multi-line
string can be given
like this""")
```

Strings have similar operations than lists, except that they are *immutable* like tuples. They also have their own operations:

```
S = "text"
print(S[2])           # "x"
print(S[1:-1])        # "ex"
print("a"*10 + "bc")  # "aaaaaaaaaabc"
print(S.replace("t", "T"))         # "TexT" (but S didn't change)
print("abc,d,e,,fg".split(","))    # ["abc", "d", "e", "", "fg"]
print(",".join(["a","bc","def"]))  # "a,bc,def"
```

A *generator* is an abstract sequence which generates its elements one by one, but it doesn't store them all at once in the memory (unlike e.g. a list). An example is `range()`, used in the `for` loops. For example, `range(1000000)` doesn't store all numbers from 0 to 999999 in memory, only one at a time. (If we really want to, we can force Python to store all these numbers by converting the generator to a list: `list(range(1000000))`.)

A generator can be defined similarly to a function, but instead of using `return` (which ends the function immediately), the `yield` command must be used, which gives the values one by one. For example, we can define a generator similar to `range(a,b)`:

```
def myrange(a, b):
   i = a
   while i < b:
      yield i
      i += 1
```

Then:

```
for i in myrange(2, 10): print(i*i)
```

Since a generator doesn't store all the elements, it can even be infinite:

```
def squares():
   i = 1
   while True:   # infinite loop
      yield i*i
      i += 1
```

Of course, the loop that uses it must then terminate itself:

```
for x in squares():
   if x > 1000: break
   print(x)
```

3 SageMath basics

The syntax of Sage is almost identical to Python. The most important syntactical difference is that exponentiation, which is denoted by two asterisks (`**`) in Python, e.g. `2**3 == 8`, is written in Sage using a caret (`^`), e.g. `2^3 == 8`. (Note: In Python, `^` is the "bitwise exclusive or" (XOR) operator, for which Sage uses double `^^`.)

When we start Sage, it works as an interpreter, i.e. it expects the commands one by one, and gives the result immediately. It can be used as a simple calculator, for example:

```
sage: 1+1
2
sage: 2^10
1024
sage: factorial(33)
8683317618811886495518194401280000000
```

Here '`sage:`' is the *prompt*, i.e. we can write the command after it.

We can refer to the previous result by an underscore (`_`), and to the one before that by double underscores (`__`) (this goes up to three underscores: `___`). For example:

```
sage: 2^3
8
sage: _ + 1
9
sage: (_ + 1) * __
80
```

For more complex programs, it's better to write it to a separate file, and load it to Sage using the `load()` function. Sage programs have the file extension `.sage`. (You can use any text editor to create the Sage file. If it doesn't support Sage syntax highlighting, you can select Python's.)

For example if `new.sage` contains the following:

```
L = [1,2,5,10]
for n in L:
   print(n, 2^n, factorial(n))
```

then we can read it like this:

```
sage: load("directory/new.sage")
1 2 1
2 4 2
5 32 120
10 1024 3628800
```

(The path in `load()` must be given relative to the current directory, i.e. from which we started Sage – or as an absolute path. The current directory can be queried by the '`pwd`' command.)

In Sage – unlike in Python – information about a function can be queried by a question mark (`?`), for example (you can quit from the help by pressing `q`):

```
sage: sqrt?
sage: solve?
```

3.1 Symbolic calculation

Sage calculates *symbolically* by default. What does it mean? For example:

```
sage: 2/3
2/3
```

This doesn't seem too useful: it only repeated the query. But not because it can't divide. We can try more:

```
sage: 666/999
2/3
sage: 6/3
2
sage: 2/3 + 1/2
7/6
```

So it calculated everything, and gave the result in the simplest form.

But why didn't it give for `2/3` what most common calculators would give (and also Python): `0.6666666666666666`? Because it's not the exact value of 2/3. The exact value is 2/3, which can't be expressed any simpler than that. The exact value's decimal representation contains infinitely many 6's, which cannot be displayed by any calculator. Therefore, most common calculators use approximations. This is known as *numerical* calculation, as opposed to symbolic (or *exact*) calculation, which Sage does by default. The drawback of numerical calculation is that the rounding errors may add up, and the result may be far from the exact value.

More examples for symbolic calculation in Sage (`sqrt(x)` $= \sqrt{x}$):

```
sage: sqrt(2)
sqrt(2)
sage: sqrt(16)
4
sage: sqrt(8)
2*sqrt(2)
sage: sqrt(2)*sqrt(8)
4
sage: sin(pi/3)
1/2*sqrt(3)
sage: log(e^3)
3
```

But Sage *can* calculate numerically if we ask it to, and for arbitrary precision:

```
sage: N(sqrt(2))
1.41421356237310
sage: N(sqrt(2), digits=80)
1.4142135623730950488016887242096980785696718753769480731766797379907324784621070
```

The result will also be numerical if the input is already a floating-point value:

```
sage: sqrt(2.)
1.41421356237310
```

But symbolic calculation means more than just exact calculation with numbers. Sage can also calculate symbolically with letters, for example:

```
sage: (x+1)^2
(x + 1)^2
```

Again, it didn't do anything, but now it's because we didn't say what to do with it. For example, we can expand a product into a sum, or we can factor a sum to get a product:

```
sage: expand((x+1)^2)
x^2 + 2*x + 1
sage: expand((x+1/x)^8)
x^8 + 8*x^6 + 28*x^4 + 56*x^2 + 56/x^2 + 28/x^4 + 8/x^6 + 1/x^8 + 70
sage: factor(x^4 - 1)
(x^2 + 1)*(x + 1)*(x - 1)
```

In Sage, `x` is a predefined *symbolic variable*. We can also define other symbolic variables using the `var()` function:

```
sage: var("a,b")
(a, b)
sage: expand(a^2*(a-b)^3)
a^5 - 3*a^4*b + 3*a^3*b^2 - a^2*b^3
```

(If we don't want to get an answer, we can suppress it by adding a semicolon after the statement, e.g. '`var("a,b");`'.)

Important: don't confuse symbolic variables with ordinary (Python-like) variables, because variables always have an actual value (e.g. `5`), while the symbolic variables refer to the symbol itself (e.g. `x`):

```
sage: n = 5      # n is a variable
sage: n^2
25
sage: type(n)
<class 'sage.rings.integer.Integer'>
sage: var("n");  # n is a symbolic variable
sage: n^2
n^2
sage: type(n)
<class 'sage.symbolic.expression.Expression'>
```

(Sidenote: as we can see, integers have different type in Sage (`Integer`) than in Python (`int`). Sage's types will be discussed later.)

And don't confuse symbolic variables with *symbolic constants* either: `pi`, `e` etc., which have specific mathematical values, even though they can also be manipulated symbolically:

```
sage: x/4, pi/4     # so far they look the same
(1/4*x, 1/4*pi)
sage: tan(x/4), tan(pi/4)    # but here comes the difference
(tan(1/4*x), 1)
sage: N(e)   # symbolic constants have a numerical value
2.71828182845905
```

```
sage: N(x)   # symbolic variables don't
[...]
TypeError: cannot evaluate symbolic expression numerically
sage: i^2
-1
sage: e^(i*pi) + 1   # Euler's identity
0
sage: N(golden_ratio)
1.61803398874989
sage: bool(golden_ratio == (sqrt(5) + 1)/2)
True
```

If we overwrite a predefined symbolic constant, we can restore it by the `reset()` function:

```
sage: log(e)
1
sage: var("a,b,c,d,e");  # oops!
sage: log(e)
log(e)
sage: reset("e")
sage: log(e)
1
```

3.2 Maths in Sage

The following common mathematical functions are available in Sage:

`abs(x)`	$\lvert x\rvert$
`conjugate(x)`	$\overline{x}$
`factorial(n)`	$n!$
`binomial(n,k)`	$\binom{n}{k}$
`sqrt(x)`	$\sqrt{x}$
`sin(x)`	$\sin x$
`cos(x)`	$\cos x$
`tan(x)`	$\operatorname{tg} x$
`log(x,a)`	$\log_a x$
`log(x)`	$\ln x = \log_e x$
`exp(x)`	e^x
`floor(x)`	$\lfloor x \rfloor$
`ceil(x)`	$\lceil x \rceil$

```
sage: sqrt(-4)
2*I
sage: conjugate(2 + 3*i)
-3*I + 2
sage: binomial(5, 2)
10
sage: var("n"); binomial(n, 2)
1/2*(n - 1)*n
sage: exp(2*log(x))
x^2
sage: floor(57/10)
5
sage: ceil(57/10)
6
```

As we saw earlier, Sage can do various manipulations on mathematical expressions (e.g. `expand()`). Some operations can be given in two forms: as a regular function, e.g. `expand(...)`, or as a member function, e.g. `(...).expand()`. (This is also true for many other functions, e.g. `4.sqrt() == 2`.)

Let's see a few more operations on expressions (again, '_' means the previous result):

```
sage: var("x,y,n");
sage: ((x+y)^2).expand()
x^2 + 2*x*y + y^2
sage: _.factor()
(x + y)^2
sage: ((x^2 - y^2) / (x - y)).simplify_full()
x + y
sage: (cos(x)^2 + sin(x)^2).simplify_full()
1
sage: (factorial(n+3) / factorial(n)).simplify_full().factor()
(n + 3)*(n + 2)*(n + 1)
sage: f = x^2 + y*x + 2
sage: f.subs(x = 2)
2*y + 6
sage: f.subs(x = sqrt(x))
sqrt(x)*y + x + 2
sage: f.subs(x = y, y = x)
x*y + y^2 + 2
sage: (x*y - 2*x^2 + x - y + x^2*y).collect(x)
x^2*(y - 2) + x*(y + 1) - y
sage: _.collect(y)
-2*x^2 + (x^2 + x - 1)*y + x
```

Sage can do summation using the `sum()` function. It has two versions. The simpler one (inherited from Python) just expects a list (or another sequence):

```
sage: sum([10,20,30])
60
sage: sum(range(10))
45
```

The other version works like the mathematical summation ($\sum$). It expects four arguments: `sum(<formula>, <variable>, <from>, <to>)`. For example:

```
sage: var("n,k");
sage: sum(k^2, k, 1, 10)
385
sage: sum(k, k, 1, n).factor()
1/2*(n + 1)*n
sage: # oo == Infinity:
sage: sum(1/2^n, n, 0, oo)
2
sage: # product works similarly:
sage: product(k^2, k, 1, n)
factorial(n)^2
```

$$\sum_{k=1}^{10} k^2 = 1^2 + 2^2 + \ldots + 10^2 = 385$$

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

$$\sum_{n=0}^{\infty} \frac{1}{2^n} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots = 2$$

$$\prod_{k=1}^{n} k^2 = 1^2 \cdot 2^2 \cdot 3^2 \cdot \ldots \cdot n^2 = (n!)^2$$

Equations can be solved symbolically using the `solve()` function:

```
sage: var("x,y");
sage: solve(x^3 + 3*x^2 == 2*x + 2, x)
[x == -sqrt(2) - 2, x == sqrt(2) - 2, x == 1]
sage: solve([x + y == 6, x - y == 4], x, y)
[[x == 5, y == 1]]
sage: solve(2*x + y^2 == 6, x)
[x == -1/2*y^2 + 3]
sage: solve(2*x + y^2 == 6, y)
[y == -sqrt(-2*x + 6), y == sqrt(-2*x + 6)]
```

Variables can be constrained by assumptions using the `assume()` function:

```
sage: solve(x^3 == 1, x)
[x == 1/2*I*sqrt(3) - 1/2, x == -1/2*I*sqrt(3) - 1/2, x == 1]
sage: assume(x, "real")
sage: solve(x^3 == 1, x)
[x == 1]
sage: forget()   # delete all assume()'s, i.e. x can be anything again
sage: sqrt(x^2)
sqrt(x^2)
sage: assume(x, "real")
sage: sqrt(x^2)
abs(x)
sage: assume(x > 0)
sage: sqrt(x^2)
x
sage: forget()
sage: var("x,y");
sage: # Infinitely many solutions will be given paremetrically:
sage: solve(x^2 + y^2 == 5, x, y)
[[x == r1, y == sqrt(-r1^2 + 5)], [x == r2, y == -sqrt(-r2^2 + 5)]]
sage: # But with integers, there are only finitely many solutions:
sage: assume(x, y, "integer")
sage: solve(x^2 + y^2 == 5, x, y)
[(2, -1), (1, 2), (-1, -2), (2, 1), (-2, -1), (-2, 1), (1, -2), (-1, 2)]
```

Not all equations can be solved symbolically. For example, there is no general formula for quintic (degree-5) equations, so Sage can't solve them either:

```
sage: solve(x^5 + x + 3, x)  # "== 0" can be omitted
[0 == x^5 + x + 3]
```

But they can be solved numerically, using the `roots()` member function, and giving the number set, e.g. $\mathbb{R}$ = `RR`, $\mathbb{C}$ = `CC` (the second number in the result is the multiplicity, i.e. `1` = single root):

```
sage: (x^5 + x + 3).roots(x, ring=RR)
[(-1.13299756588507, 1)]
```

Sage is also strong in calculus (limits, differentiation and integration):

```
sage: var("n");
sage: limit(2*n/(3*n + 1), n = oo)
2/3
sage: limit(1/(x-1), x = 1, dir='-')
-Infinity
sage: diff(x^3 + sin(2*x), x)
3*x^2 + 2*cos(2*x)
sage: diff(x^3 + sin(2*x), x, 2)
6*x - 4*sin(2*x)
sage: integral(3*x^2 + 2*cos(2*x), x)
x^3 + sin(2*x)
sage: integral(sin(x), x, 0, pi)
2
```

$$\lim_{n\to\infty} \frac{2n}{3n+1} = \frac{2}{3}$$

$$\lim_{x\to 1^-} \frac{1}{x-1} = -\infty$$

$$\big(x^3 + \sin(2x)\big)' = 3x^2 + 2\cos(2x)$$

$$\big(x^3 + \sin(2x)\big)'' = 6x - 4\cos(2x)$$

$$\int \big(3x^2 + 2\cos(2x)\big)\, dx = x^3 + \sin(2x) + C$$

$$\int_0^{\pi} \sin x\, dx = 2$$

Vectors can be created using the `vector()` function:

```
sage: v = vector([1,3,-2])
sage: v
(1, 3, -2)
sage: v[2]
-2
sage: w = vector([2,1,0])
sage: v + 2*w
(5, 5, -2)
sage: v*w
5
sage: v.norm(2)  # length (Euclidean norm)
sqrt(14)
```

And `matrix()` creates a matrix:

```
sage: M = matrix([[3,1,-4], [2,5,6], [1,4,8]])
sage: M
[ 3  1 -4]
[ 2  5  6]
[ 1  4  8]
sage: M[1][2]   # second row, third element
6
sage: M.det()   # determinant
26
sage: M.inverse()
[  8/13 -12/13      1]
[ -5/13  14/13     -1]
[  3/26 -11/26    1/2]
```

```
sage: M.inverse() * M == matrix.identity(3)
True
sage: matrix([[1,-1,-1], [1,1,0], [3,0,1]]).eigenvalues()
[1, 1 - 2*I, 1 + 2*I]
```

And of course, matrices and vectors work symbolically too, for example we can manipulate 2×2 matrices in a general form:

```
sage: var("a,b,c,d");
sage: M = matrix([[a,b], [c,d]])
sage: M
[a b]
[c d]
sage: M.det()
-b*c + a*d
sage: M^2
[a^2 + b*c a*b + b*d]
[a*c + c*d b*c + d^2]
sage: (M.inverse() * M).simplify_full()
[1 0]
[0 1]
sage: var("x,y");
sage: v = vector([x,y])
sage: M * v
(a*x + b*y, c*x + d*y)
sage: v.norm(2)
sqrt(abs(x)^2 + abs(y)^2)
```

Systems of linear equations can be solved in multiple ways:

```
sage: A = matrix([[0,1,-3], [4,5,-2], [2,3,-1]]); b = vector([-5,10,7])
sage: A.solve_right(b)  # A*x == b  ("right": x multiplies from the right)
(-1, 4, 3)
sage: A*_ == b
True
sage: A.inverse() * b   # less efficient
(-1, 4, 3)
sage: var("x,y,z");
sage: solve([y - 3*z == -5, 4*x + 5*y - 2*z == 10, 2*x + 3*y - z == 7], x,y,z)
[[x == -1, y == 4, z == 3]]
```

4 Author

These lecture notes were written by Uray M. János, partially using Nagy Ádám's notes in Hungarian:
https://compalg.elte.gitlab-pages.hu/dimoa-web/tematika/dimoa
Please report errors and suggestions here: uray.janos@inf.elte.hu.